1.5 Building Up Data with Comprehensions

To wrap up our introduction to data in Python, we're going to learn about one last kind of expression that allows to build up and transform large collections of data in Python.

From set builder notation to set comprehensions

Recall *set builder notation*, which is a concise way of defining a mathematical set by specifying the values of the elements in terms of a larger domain. For example, suppose we have a set $S = \{1, 2, 3, 4, 5\}$. We can express a set of squares of the elements of S:

$$\{x^2 \mid x \in S\}.$$

It turns out that this notation translates naturally to Python! To start, let's go into the Python Console and create a variable that refers to a set of numbers:

```
>>> numbers = {1, 2, 3, 4, 5}
```

Now, we introduce a new kind of expression called a **set comprehension**, which has the following syntax:[1]

[1] Careful with this: even though set comprehensions also use curly braces, they are *not* the same as set literals. We aren't writing out the individual elements separated by commas.

```
{ <expr> for <variable> in <collection> }
```

Evaluating a set comprehension is done by taking the `<expr>` and evaluating it once for each value in `<collection>` assigned to the `<variable>`. This is exactly analogous to set builder notation, except using `for` instead of $\mid$ and `in` instead of $\in$. Here's how we can repeat our initial example in Python using a set comprehension:

```
>>> {x ** 2 for x in numbers}
{1, 4, 9, 16, 25}
```

Pretty cool, eh? If you aren't sure exactly what happened here, it's useful to write out the expanded form of the set comprehension:

```
   {x ** 2 for x in numbers}
== {1 ** 2, 2 ** 2, 3 ** 2, 4 ** 2, 5 ** 2}  # Replacing x with 1, 2, 3, 4,
        and 5.
```

It goes even further—we can use set comprehensions with a Python list as well.

```
>>> {x ** 2 for x in [1, 2, 3, 4, 5]}
{1, 4, 9, 16, 25}
```

In fact, as we'll see later in this course, set comprehensions can be used with any "collection" data type in Python, not just sets and lists.

List and dictionary comprehensions

Even though set comprehensions draw their inspiration from set builder notation in mathematics, Python has extended them to other data types.

A **list comprehension** is very similar to a set comprehension, except its syntax uses square brackets instead of curly braces:

```
[ <expr> for <variable> in <collection> ]
```

Once again, `<collection>` can be a set or a list:

```
>>> [x + 4 for x in {10, 20, 30}]
[14, 24, 34]
>>> [x * 3 for x in [100, 200, 300]]
[300, 600, 900]
```

One word of warning: because sets are unordered but lists are ordered, you should *not* assume a particular ordering of the elements when a list comprehension generates elements from a set—the results can be unexpected!

```
>>> [x for x in {20, 10, 30}]
[10, 20, 30]
```

A **dictionary comprehension** is again similar to a set comprehension, but specifies both an expression to generate keys and an expression to generate their associated values:

```
{ <key_expr> : <value_expr> for <variable> in <collection> }
```

Out of all three comprehension types, dictionary comprehensions are the most complex, because the left-hand side (before the `for`) consists of two expressions instead of one. Here is one example of a dictionary comprehension that creates a "table of values" for the function $f(x) = x^2 + 1$.

```
>>> {x : x ** 2 + 1 for x in {1, 2, 3, 4, 5}}
{1: 2, 2: 5, 3: 10, 4: 17, 5: 26}
```

Comprehensions with multiple variables

Our last example in this section will be to illustrate how multiple variables are used within the same comprehension expression. First, recall how we defined the *Cartesian product* of two sets using set builder notation:

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}.$$

In this expression, the expression (x, y) is evaluated once for every possible combination of elements x of A and elements y of B.

The same holds for set, list, and dictionary comprehensions. We can specify additional variables in a comprehension by adding extra `for <variable> in <collection>` clauses to the comprehension. For example, if we define the following sets:

```
>>> nums1 = {1, 2, 3}
>>> nums2 = {10, 20, 30}
```

then we can calculate their Cartesian product using the following set comprehension:[2]

[2] Remember, sets are unordered! Don't get hung up on the unusual order in the output.

```
>>> {(x, y) for x in nums1 for y in nums2}
{(3, 30), (2, 20), (2, 10), (1, 30), (3, 20), (1, 20), (3, 10), (1, 10), (2,
         30)}
```

In general, if we have a comprehension with clauses `for v1 in collection1`, `for v2 in collection2`, etc., then the comprehension's inner expression is evaluated *once for each combination of values for the variables*. This illustrates yet another pretty impressive power of Python: the ability to combine different collections of data together in a short amount of code.